Python Tutorial

Projekt Collaborative Writing

Hochschule Kaiserslautern

Inhaltsverzeichnis

Kapitel 1

Grundlagen

1.1 Elementare Datentypen

Ähnlich wie bei Java und C oder C++, gibt es auch in Python Variablen. Allerdings gibt es dabei immense Unterschiede zu den anderen Programmiersprachen, weshalb sich ein genauerer Blick auf die einzelnen Datentypen in jedem Fall lohnt. Bei vielen bekannten Sprachen wird einer Variablen ein bestimmter Datentyp zugeordnet (deklariert). Der Datentyp kann darauf folgend zur Laufzeit nicht wieder geändert werden, der Wert innerhalb des Datentyps allerdings schon. So lassen sich in eine Variable des Typ Integer beispielsweise keine String-Werte speichern. In Python hingegen, ist dies ohne weiteres möglich. Hier wird nämlich gänzlich auf eine explizite Typdeklaration verzichtet. Zeigt eine Variable beispielsweise auf eine ganze Zahl, so wird diese als ein Objekt vom Typ Integer interpretiert. Allerdings kann man sie im nächsten Schritt einfach auf ein String-Objekt zeigen lassen. Soviel zum allgemeinen Unterschied zu den anderen Programmiersprachen. Betrachten wir nun die Datentypen etwas genauer.

1.1.1 Zahlenoperatoren

Da in Python auf Typdeklarationen verzichtet wird, muss dieser nicht beim Anlegen der Variable berücksichtigt werden. Wird eine ganze Zahl (Integer)benötigt, kann diese falls nötig auch in eine Gleitkommazahl (float) umgewandelt werden, ohne groß etwas am Code zu ändern. Python deklariert im Hintergrund selbst und spart so unnötige Komplexitäten und Fehlerquellen (Beispiel in 1.1).

Listing 1.1: Zahlenoperatoren

```
i = 42
type(i)
// Ausgabe: <class 'int'>
i = 42.22
type(i)
// Ausgabe: <class 'float'>
```

Boolean

Boolean gibt an, ob ein Statement true oder false ist. Dadurch lassen sich Fallunterscheidungen oder Abfragen ermöglichen (Beispiel in Listing 1.2).

Listing 1.2: Boolean

```
i = True
i
// Ausgabe: True
```

String

Der String ist eine Zeichenkette, also eine Aneinanderreihung von verschiedenen Zeichen. Dazu zählen Wörter, aber auch beispielsweise Hexadezimal-Codes oder E-Mail Adressen.

Wie in den meisten objektorientierten Programmiersprachen, lassen sich auch in Python die einzelnen Zeichen eines Strings abrufen in dem der dazugehörige Index abgerufen wird.

Wie in Listing 1.3 kann die Länge des gesamten Strings durch einfache Abfrage angezeigt werden.

Listing 1.3: Strings

```
i = "Python"
print (i)
// Ausgabe: Python

print(i[0])
// Ausgabe: P

print(len(i))
// Ausgabe: 6
```

1.2 Collections

In Python 3 existieren nativ die vier Datenstrukturen List, Tuple, Set und Dictionary, welche im Folgenden vorgestellt werden.

1.2.1 List

Die Datenstruktur List bietet einen geordneten und veränderbaren Behälter für Python-Objekte, der Duplikate von Elementen erlaubt. Da eine List immer sortiert ist, können einzelne Elemente aus der Datenstruktur über den entsprechenden Index ausgewählt und verändert werden. Python unterstützt intern keine Arrays, alternativ hierzu kann eine List verwendet werden.

Eine List kann wie folgt initialisiert werden:

```
# listings/ListInit.py
# Initialisierung einer List

liste = [1, 2, 3]

# oder
liste = list((1, 2, 3))
```

Dabei kann sie jegliche Art von Objekten beinhalten; der Datentyp spielt hierbei keine Rolle.

Beispiel:

```
# listings/ListDataType.py
# Objekte mit unterschiedlichen Datentypen in einer List

liste = [1, "hallo", 2.3, (5, 6), [22, 23, 24], 'a']
```

Im Gegensatz zu Java und C++ muss der Programmierer darauf achten und sicherstellen, dass die Datenstruktur mit Werten des entsprechenden Datentyps befüllt wird, um Fehler aufgrund unterschiedlicher Datentypen zu vermeiden.

Der Inhalt einer List kann über die `print`-Methode ausgegeben werden. Im folgenden Beispiel werden verschiedene Elemente der List auf der Konsole ausgegeben. Wird die List als Parameter gewählt, wird der Inhalt ausgegeben.

```
# listings/ListPrint.py
# Ausgabe des Inhalts einer List auf der Konsole

liste = [1, 2, 3]
print(liste)
```

Wie zuvor erwähnt, ähnelt die Verwaltung einer List der eines Arrays aus Java oder C++. Durch die Verwendung eines Index können einzelne Elemente ausgewählt oder verändert werden.

```
# listings/ListIndex.py
# Beispiel fuer das Ueberschreiben und Ausgeben eines
# einzelnen Elements einer List

liste = [1, 2, 3]
print(liste[1])
liste[1] = 4
print(liste)
```

Python erlaubt die Nutzung von negativen Indizes. Mit diesen kann der Inhalt der List in umgekehrter Reihenfolge ausgegeben werden. Ein Index von `-1` wird dem letzten Element der List zugeordnet, `-2` dem vorletzten.

```
# listings/ListNegativeIndex.py
# Beispiel fuer die Verwendung eines negativen Index

liste = [1, 2, 3]
```

```
print(liste[-1])
print(liste[-2])
```

In Python existiert für die Datenstruktur List keine Methode, die mit `contains` in Java oder der `find` aus C++ vergleichbar ist. Stattdessen stehen die Membership Operatoren `in` oder `not in` zur Verfügung, welche auf eine beliebige Sequenz oder die hier beschriebenen Collections angewendet, Auskunft darüber gibt, ob das spezifizierte Element darin enthalten ist.

```
# listings/ListInOperator.py
# Verwendung des in-Operators

liste = [1, 2, 3]

print(2 in liste)

if 2 in liste:
    print("Gefunden!")
else:
    print("Nicht gefunden!")
```

Der Python Interpreter stellt nativ einige Funktionen zur Verfügung. Eine davon ist die `len`-Methode, welche die Anzahl an Elementen in einem Objekt liefert.

```
# listings/ListLen
# Ausgaben der Anzahl der Elemente in einer List

liste = [1, 2, 3]

print(len(liste))
```

Das `del`-Statement erlaubt das Löschen einzelner Elemente oder der gesamten List.

```
# listings/ListDelete.py
# Beispiel zur Verwendung des del-Operators

liste = [1, 2, 3]
print(liste)

del liste[1]
print(liste)
```

```
del liste

print(liste)  # ERROR, da die Liste nicht mehr existiert!
```

Methoden einer List

append(): Fügt am Ende der List ein Objekt hinzu.

```
# listings/ListAppend.py
# Verwendung der append-Methode

liste = [1, 2, 3]
print(liste)
liste.append(4)
print(liste)
```

clear(): Entfernt sämtliche Objekte aus der List.

```
# listings/ListClear.py
# Verwendung der clear-Methode

liste = [1, 2, 3]
liste.clear()
print(liste)
```

copy(): Liefert eine Kopie der List.

```
# listings/ListCopy.py
# Verwendung der copy-Methode

liste = [1, 2, 3]
duplicate = liste.copy()
print(duplicate)
```

count(): Liefert die Anzahl des spezifizierten Objekts in der List.

```
# listings/ListCount.py
# Verwendung der count-Methode

liste = [1, 2, 3, 2, 2]
print(liste.count(2))
```

extend(): Fügt der `liste1` den Inhalt der `liste2` am Ende hinzu.

```
# listings/ListExtend.py
# Verwendung der extend-Methode

liste1 = [1, 2, 3]
liste2 = [4, 5, 6]
liste1.extend(liste2)
print(liste1)
```

index(): Liefert den Index der Position, an der sich das erste spezifizierte Objekt in der List befindet.

```
# listings/ListIndexMethode.py
# Verwendung der index-Methode

liste = [1, 2, 3, 2, 2]
print(liste.index(2))
```

insert(): Fügt ein Objekt an der gewählten Position der List hinzu.

```
# listings/ListInsert.py
# Verwendung der insert-Methode

liste = ["1", "2", "3"]
liste.insert(1, "4")
print(liste)
```

pop(): Entfernt das Objekt, das sich an der durch den Index spezifizierten Position befindet.

```
# listings/ListPop.py
# Verwendung der pop-Methode

liste = [1, 2, 3]
liste.pop(1)
print(liste)
```

remove(): Entfernt das erste Objekt der List, das der Spezifikation entspricht.

```
# listings/ListRemove.py
# Verwendung der remove-Methode

liste = [1, 2, 3, 2]
```

```
liste.remove(2)
print(liste)
```

reverse(): Invertiert die Folge der Objekte in der List.

```
# listings/ListReverse.py
# Verwendung der reverse-Methode

liste = [1, 2, 3]
liste.reverse()
print(liste)
```

sort(): Sortiert die List.

```
# listings/ListSort.py
# Verwendung der sort-Methode

# Lexikographisches Sortieren
liste = ["b", "c", "a"]
print(liste)
liste.sort()
print(liste)

# Sortieren nach Zahlenwert
liste = [6, 1, 2, 3, 4, 5]
print(liste)
liste.sort()
print(liste)

# Umkehren der Sortierreihenfolge
liste.sort(reverse=True)
print(liste)

# Sortieren nach der Laenge einzelner Objekte
liste = ["aa", "aaa", "a"]


def sortFunc(x):
    return len(x)


liste.sort(key=sortFunc)
print(liste)
```

```
# Umkehren der Sortierreihenfolge
liste.sort(reverse=True, key=sortFunc)
print(liste)
```

1.2.2 Tuple

Ein Tuple stellt einen geordneten und unveränderbaren Behälter für Python-Objekte dar. Dieser erlaubt, wie eine List, Duplikate und den Zugriff auf einzelne Elemente über einen Index. Tuple sind Datenstrukturen, die ausschließlich gelesen werden können.

Ein Tuple wird mit folgender Syntax erzeugt:

```
# listings/TupleInit.py
# Die Initialisierung eines Tuples

tupel = (1, 2, 3)

# oder
tupel = tuple((1, 2, 3))
```

Es ist möglich, leere Tuple zu erzeugen. Wie zuvor erwähnt, ist deren Inhalt unveränderlich.

Arbeiten mit einem Tuple

Der Inhalt eines Tuple kann, analog zur List, auf der Konsole ausgegeben werden. Das Zuweisen eines neuen Objekts mittels Index führt im Gegensatz zur List zu einem Fehler.

```
# listings/TupleIndex.py
# Zuweisung eines Objekts ueber den Index

tupel = (1, 2, 3)
tupel[0] = 4  # ERROR
```

Die Verwendung der Operatoren `in` und `not in` ist, wie die `len()`-Methode, analog zur List-Datenstruktur.

```
# listings/TupleInLen.py
```

```
# Verwendung des "in"-Operators

tupel = (1, 2, 3)

print(2 in tupel)
print(len(tupel))
```

Das `del`-Statement erlaubt das Löschen des Tuple. Aufgrund der Unveränderbarkeit der Datenstruktur können keine einzelnen Elemente entfernt werden.

```
# listings/TupleDelete.py
# Verwendung des del-Statements

tupel = (1, 2, 3)

del tupel
```

Methoden eines Tuple

count(): Liefert die Anzahl des gewählten Werts in einem Tuple.

```
# listings/TupleCount.py
# Verwendung der count-Methode

tupel = (1, 2, 4, 3, 2, 2)
print(tupel.count(2))
```

index(): Liefert die Position des ersten Werts, der mit dem spezifizierten Wert übereinstimmt.

```
# listings/TupleIndexMethode.py
# Verwendung der index-Methode

tupel = (1, 2, 3, 2)
print(tupel.index(2))
```

1.2.3 Set

Ein Set ist durch das Hinzufügen oder Entfernen von Objekten veränderbar und erlaubt keine Duplikate. Das Initialisieren mit mehrfach identischen

Werten führt nicht zu einem Fehler, jedoch werden die überzähligen Werte aus dem Set entfernt. Die enthaltenen Elemente sind unveränderlich. Zudem ist die Datenstruktur ungeordnet, weshalb nicht auf einzelne Objekte mittels Index zugegriffen werden kann.

Ein Datenbehälter vom Typ Set kann mit folgender Syntax erzeugt werden:

```
# listings/SetInit.py
# Die Initialisierung eines Sets

set1 = {1, 2, 3}

# oder
set1 = set((1, 2, 3))
```

Arbeiten mit Sets

Bei der Ausgabe eines Set auf der Konsole ist die Reihenfolge der Elemente nicht garantiert.

Die Syntax für die Ausgabe auf der Konsole ist analog zur List. Die Verwendung eines Index ist nicht erlaubt und führt zu einem Fehler.

```
# listings/SetPrint.py
# Ausgabe des Inhalts eines Set auf der Konsole

set1 = {1, 2, 3}
print(set1)
for x in set1:
    print(x)
print(set1[0])  # ERROR
set1[1] = 4  # ERROR
```

Methoden eines Sets

add(): Fügt dem Set ein Objekt hinzu.

```
# listings/SetAdd.py
# Verwendung der add-Methode

set1 = {1, 2, 3}
print(set1)
```

```
set1.add(4)
print(set1)
```

clear(): Entfernt alle Elemente aus dem Set.

```
# listings/SetClear.py
# Verwendung der clear-Methode

set1 = {1, 2, 3}
print(set1)
set1.clear()
print(set1)
```

copy(): Liefert eine Kopie des Sets.

```
# listings/SetCopy.py
# Verwendung der copy-Methode

set1 = {1, 2, 3}
x = set1.copy()
print(x)
```

difference(): Liefert ein Set, das diejenigen Elemente enthält, die ausschließlich in `setX` vorkommen. Alle Element, die mit denen von `setY` übereinstimmen, werden aus dem ersten entfernt. Alternativ ist dies auch über den Operator - möglich.

```
# listings/SetDifference.py
# Verwendung der difference-Methode


set1 = {1, 2, 3}
set2 = {3, 8, 4}
x = set1.difference(set2)
print(x)

# oder
y = set1 - set2
print(y)
```

difference_update(): Entfernt diejenigen Elemente aus dem ersten Set, die mit denen aus dem zweiten übereinstimmen.

```
# listings/SetDifferenceUpdate.py
# Verwendung der difference_update-Methode


setX = {1, 2, 3}
setY = {3, 8, 4}
setX.difference_update(setY)
print(setX)
```

discard(): Entfernt das gewählte Element aus dem Set. Duplikate werden ebenfalls entfernt.

```
# listings/SetDiscard.py
# Verwendung der discard-Methode

set1 = {1, 2, 4, 3, 4}
set1.discard(4)
print(set1)
```

intersection(): Liefert ein Set mit der Schnittmenge zweier Sets. Alternativ ist dies auch mit der Angabe des `&`-Operators möglich.

```
# listings/SetIntersection.py
# Verwendung der intersection-Methode

setX = {1, 2, 3, 4, 5}
setY = {3, 4, 9, 5, 8, 7}
print(setX & setY)
print(setX.intersection(setY))
```

intersection_update(): Entfernt alle Elemente, die sich nicht in der Schnittmenge beider Sets befinden.

```
# listings/SetIntersectionUpdate.py
# Verwendung der intersection_update-Methode

setX = {1, 2, 3, 4, 5}
setY = {3, 4, 9, 5, 8, 7}
setX.intersection_update(setY)
print(setX)
```

isdisjoint(): Gibt Auskunft darüber, ob zwei Sets eine Schnittmenge besitzen. Liefert `True`, wenn kein Element des ersten Sets im zweiten enthalten ist.

```
# listings/SetIsDisJoint.py
# Verwendung der isdisjoint-Methode

setX = {1, 2, 3, 4, 5}
setY = {6, 7, 8, 9, 10}
print(setX.isdisjoint(setY))  # Liefert True

setX = {1, 2, 3, 4, 5}
setY = {3, 4, 9, 5, 8, 7}
print(setX.isdisjoint(setY))  # Liefert False
```

issubset(): Gibt an, ob das gewählte Set eine Teilmenge enthält, die exakt dem ersten Set entspricht. Alternativ kann das Zeichen < verwendet werden.

```
# listings/SetIsSubSet.py
# Verwendung der issubset-Methode

setX = {3, 4, 5}
setY = {3, 4, 9, 5, 8, 7}
print(setX.issubset(setY))
print(setX < setY)
```

pop(): Entfernt ein beliebiges Element aus dem Set. Sollte das Set leer sein, wird ein Fehler generiert.

```
# listings/SetPop.py
# Verwendung der pop-Methode

set1 = {1, 2, 3}
set1.pop()
print(set1)
```

remove(): Entfernt das gewählte Element aus dem Set. Sollte das gewählte Element nicht in dem Set enthalten sein, wird ein Fehler angezeigt.

```
# listings/SetRemove.py
# Verwendung der remove-Methode

set1 = {1, 2, 3}
print(set1)
set1.remove(3)
print(set1)
```

symmetric_difference(): Liefert ein Set, das die Vereinigung zweier Sets ohne deren Schnittmenge enthält.

```
# listings/SetSymDiff.py
# Verwendung der symetric_difference-Methode

set1 = {1, 2, 3, 4, 5}
set2 = {6, 7, 8, 9, 10}
print(set1.symmetric_difference(set2))
```

symmetric_difference_update(): Vereinigt zwei Sets und entfernt deren Schnittmenge.

```
# listings/SetSymDiffUpdate.py
# Verwendung der symmetric_difference_update-Methode

set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8, 9, 10}
set1.symmetric_difference_update(set2)
print(set1)
```

union(): Liefert ein Set, das die Vereinigung zweier Sets darstellt. Duplikate werden entfernt.

```
# listings/SetUnion.py
# Verwendung der union-Methode

set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8, 9, 10}
print(set1.union(set2))
```

update(): Fügt einem Set die Items eines anderen hinzu. Duplikate werden entfernt.

```
# listings/SetUpdate.py
# Verwendung der update-Methode

set1 = {1, 2, 3, 4, 5}
set2 = {6, 7, 8, 9, 10}
set1.update(set2)
print(set1)
```

Frozenset

Im Gegensatz zu einem „normalen“ Set kann ein Frozenset nicht mehr verändert werden. Das Hinzufügen eines neuen Elements ist nicht erlaubt und führt zu einem Fehler.

```
# listings/SetFrozen.py
# Die Initialisierung eines Frozenset

set1 = frozenset([1, 2, 3, 4])
set1.add(5)  # ERROR
```

1.2.4 Dictionary

Ein Dictionary ist eine ungeordnete, veränderbare Datenstruktur, die keine Duplikate erlaubt und Schlüssel-Objekt-Paare beinhaltet. Auch beim Dictionary ist die Reihenfolge der Ausgabe nicht garantiert, denn ein Dictionary besitzt keine Ordnung.

Ein Datenbehälter vom Typ Dictionary kann mit folgender Syntax erzeugt werden:

```
# listings/DictInit.py
# Initialisierung eines Dictionary

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}

dictionary = dict(k1="v1", k2="v2", k3="v3")
```

Demnach befindet sich hinter dem Schlüssel `k1` das Objekt `v1` und analog dazu die weiteren Schlüssel-Objekt-Paare. Über den Schlüssel `k1` lässt sich auf das Objekt `v1` direkt zugreifen. Ebenso kann ein neues Objekt unter dem Schlüssel `k1` zugewiesen werden.

```
# listings/DictPrint.py
# Ausgabe des, dem Schluessel k1 zugeordneten, Objekts

dictionary = {
```

```
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}

print(dictionary["k1"])
```

Eine alternative Möglichkeit, ein Dictionary zu erstellen, ist die Methode `zip`. Mit deren Hilfe kann aus zwei separaten List-Behältern ein Dictionary generiert werden.

```
# listings/DictZip.py
# Verwendung der zip-Methode

sprache = ["englisch", "deutsch", "franzoesisch"]
laender = ["England", "Deutschland", "Frankreich"]

laendersprache = dict(zip(laender, sprache))

print(laendersprache)
```

Methoden eines Dictionary

clear(): Entfernt alle Einträge aus dem Dictionary.

```
# listings/DictClear.py
# Verwendung der clear-Methode

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary)
dictionary.clear()
print(dictionary)
```

copy(): Liefert eine Kopie des Dictionary.

```
# listings/DictCopy
# Verwendung der copy-Methode
```

```
dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
x = dictionary.copy()
print(x)
```

fromkeys(): Liefert ein Dictionary mit den angegebenen Schlüsseln und Objekten.

```
# listings/DictFromKeys.py
# Verwendung der fromkeys-Methode

x = ("k1", "k2", "k3")
y = "v"
dictionary = dict.fromkeys(x, y)
print(dictionary)
```

get(): Liefert das Objekt, das dem angegebenen Schlüssel zugeordnet ist.

```
# listings/DictGet.py
# Verwendung der get-Methode

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary.get("k1"))
```

items(): Liefert eine List mit einem Tuple für jedes Schlüssel-Objekt-Paar.

```
# listings/DictItems.py
# Verwendung der items-Methode

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary.items())
```

keys(): Liefert eine List von allen im Dictionary verwendeten Schlüsseln.

```
# listings/DictKeys.py
# Verwendung der keys-Methode

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary.keys())
```

pop(): Entfernt das Element mit dem entsprechenden Schlüssel aus dem Dictionary und liefert das Objekt zurück.

```
# listings/DictPop.py
# Verwendung der pop-Methode

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary.pop("k1"))
```

popitem(): Liefert das zuletzt hinzugefügte Schlüssel-Objekt-Paar als Tuple und entfernt es aus dem Dictionary.

```
# listings/DictPopItem.py
# Verwendung der popitem-Methode

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary.popitem())
```

setdefault(): Liefert das dem Schlüssel zugeordneten Objekt. Existiert dieser Schlüssel nicht, wird ein neues Schlüssel-Objekt-Paar mit dem angegebenen Schlüssel und Objekt angelegt.

```
# listings/DictSetDefault.py
# Verwendung der setdefault-Methode
```

```
dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
x = dictionary.setdefault("k2", "v4")
print(x)
print(dictionary)
x = dictionary.setdefault("k4", "v5")
print(x)
print(dictionary)
```

update(): Fügt dem Dictionary ein Schlüssel-Objekt-Paar hinzu.

```
# listings/DictUpdate.py
# Verwendung der update-Methode

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary)
dictionary.update({"k5": "v5"})
print(dictionary)
```

values(): Liefert eine Liste mit allen im Dictionary enthaltenen Werten.

```
# listings/DictValues.py
# Verwendung der values-Methode

dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary.values())
```

1.2.5 Zusammenfassung

In diesem Abschnitt wurde gezeigt, dass Python 3 uns mehrere Collections zur Aufbewahrung von Daten bereitstellt. Diese können je nach Datenstruk-

tur unterschiedliche Eigenschaften aufweisen. Während eine List die Daten sortiert vorhält und Duplikate zulässt, werden bei einem Set entsprechende doppelte Einträge vermieden. Betrachtet man das Set und seine Methoden genauer, ist dies der dahinter liegenden Mathematik, konkret der Mengenlehre geschuldet. Aus diesem Grund können alle gängigen mathematischen Operation auf Sets angewendet werden. Zum Schluss haben wir in diesem Abschnitt das Dictionary kennengelernt. Dieses ist ähnlich den Maps in Java. Dabei besteht ein Dictionary aus Schlüssel-Objekt-Paaren, die hinter jedem Schlüssel ein entsprechendes Objekt Mappen bzw. bereitstellen.

1.3 Kontrollstrukturen

Die Kontrollstrukturen in Python haben einen formalen Unterschied zu Java oder C++, funktional sind sie identisch. In Python werden keine geschweiften Klammern genutzt, um die Blöcke der einzelnen Abfragen abzugrenzen. Dazu genügt das Einrücken der Anweisung. Dies gilt sowohl für if-then-else-Bedingungen und Conditional Expressions, als auch für Schleifen. Im Folgenden schauen wir uns die einzelnen Strukturen im Detail und mit Beispielen an.

1.3.1 If-then-else

Die if-then-else-Struktur ermöglicht es - wie wir es bereits kennen - simple wenn-dann Abfragen zu tätigen.
Mehrere Abfragemöglichkeiten werden mit elif markiert. Vergleich hierzu Listing 1.4.

Listing 1.4: If-then-else

```
if statement1:
        print("Fall 1")
elif statement2:
        print("Fall 2")
else:
        print("Fall 3")
```

Conditional Expressions

Die Conditional Expressions (engl. bedingte Ausdrücke) stellen eine kompaktere Schreibweise als if-then-else-Bedinungen dar. Ein Beispiel ist in Listing 1.5 zu finden.

Listing 1.5: Conditional Expressions

```
// Klassisches If-Else
if wort == "start":
        x = "los"
else:
        x = halt"

// If-Else als Conditional Expression
x = ("los" if wort == "start" else "halt")
```

1.3.2 Schleifen

Python hat sowohl while- als auch for-Schleifen, welche wir uns beide im Folgenden genauer ansehen werden (vgl. Listing 1.6 und 1.7). Schleifen bestehen aus einer Anweisung und einem Kontrollblock, welcher solange durchlaufen wird, bis die Anweisung oder ein Abbruchkriterium erfüllt wurde. Schleifen die niemals ein Abbruchkriterium erfüllen und so endlos durchlaufen werden heißen Endlosschleifen. Diese führen dazu, dass der Interpreter irgendwann den Geist aufgibt und abbricht.

Listing 1.6: While-Schleife

```
// While Schleife
while Bedingung:
        Anweisungsblock
        if Bedingung:
                Anweisungsblock
                continue
        if Bedingung:
                Anweisungsblock
                break
        Anweisungsblock
```

Listing 1.7: For-Schleife

```
// For Schleife
for Variable in Objekt:
        Anweisungsblock
        if Bedingung:
                Anweisungsblock
                continue
        Anweisungsblock
        if Bedingung:
                Anweisungsblock
                break
        Anweisungsblock
```

Enums

Enums dienen in den objektorientierten Programmiersprachen zur Aufzählung von Ausdrücken einer endlichen Menge. So werden zum Beispiel Jahreszeiten, Monate oder Farben oft als Enums umgesetzt. Siehe hierzu Listing 1.8.

Listing 1.8: Enums

```
from enum import Enum
class Color(Enum):
        RED = 1
        GREEN = 2
        BLUE = 3
```

Kapitel 2

Benutzeroberflächen

Kapitel 3

Python Bibliotheken

3.1 NumPy

NumPy ist eine Python-Bibliothek für wissenschaftliches Rechnen. Sie beinhaltet unter anderem Folgendes:

- mächtige n-dimensionale Array-Objekte
- ausgeklügelte Funktionen
- Werkzeuge zur Integration von C und Fortran
- Lineare Algebra, Fouriertransformation, Erzeugung von Zufallszahlen

Um Numpy zu installieren, kann der Befehl `pip install numpy` verwendet werden.

3.1.1 Arrays

Der Array-Datentyp von NumPy heißt `numpy.ndarray`. Anders als Pythons Listentyp `list` unterstützt `numpy.ndarray` numerische Operationen auf Arrays. Die Grundrechenarten werden zwischen zwei Arrays elementweise und zwischen Array und `int/float` für alle Elemente des Arrays durchgeführt.

So kann etwa jeder Wert in einem Array mit den folgenden Anweisungen um drei erhöht werden:

```
import numpy as np
```

```
a = np.array([1,2,3])
a + 3 # [4 5 6]
```

Subtraktion, Multiplikation, Division, Ganzzahldivision und Potenzieren funktionieren analog:[1]

```
a = np.array([1,2,3])
a - 3  # [-2 -1  0]
a * 3  # [3 6 9]
a / 3  # [0.33333333 0.66666667 1.        ]
a // 3 # [0 0 1]
a ** 3 # [ 1  8 27]
```

Zwei Arrays gleicher Länge können elementweise miteinander verknüpft werden:

```
a = np.array([1,2,3])
b = np.array([4,5,6])
a + b  # [5 7 9]
a - b  # [-3 -3 -3]
a * b  # [ 4 10 18]
a / b  # [0.25 0.4  0.5 ]
a ** b # [  1  32 729]
a // b # [0 0 0]
```

Um ein Numpy-Array zu erzeugen, übergibt man ein `list`-Objekt an `np.array()`, dabei wird der in der `list` enthaltene Datentyp in einem Datentyp von Numpy konvertiert. Mit `.dtype.name` kann man den Datentyp eines Arrays herausfinden. Anders als bei `list` müssen sämtliche Elemente eines Arrays den gleichen Typ haben.

```
a = np.array([1,2,3])
a.dtype.name # 'int64'
b = np.array([1.4,2.5,3.6])
a.dtype.name # 'float64'
```

Wenn `int` und `float` gemischt übergeben werden, konvertiert Numpy in den Fließkommatyp. Wie viele Bit für einen Integer bzw. ein Float zur Verfügung stehen, ist von der Prozessorarchitektur abhängig. Moderne Computer unterstüzten in der Regel eine Größe von 64 Bit.

[1] Der Import von `numpy` wird der Übersichtlichkeit halber nachfolgend ausgelassen.

3.1.2 Konstanten und Funktionen

Es stehen für die mathematische Anwendungen auch Konstanten zur Verfügung, darunter die Folgenden mit den entsprechenden Werten und Präzisionen mit `float64`:

```
>>> np.pi
3.141592653589793
>>> np.e
2.718281828459045
>>> np.euler_gamma
0.5772156649015329
>>> np.PINF
inf
>>> np.NINF
-inf
>>> np.NAN
nan
>>> np.PZERO
0.0
>>> np.NZERO
-0.0
>>> np.NAN
nan
```

`np.NZERO` steht für die negative Darstellung der Null bei Fließkommazahlen, `np.PZERO` für die positive Dartellung.

NumPy unterstützt eine Vielzahl an mathematischen Funktionen, darunter unter anderem trigonometrische Funktionen, Rundungs-, Summations- und Multiplikationsfunktionen und Funktionen zur Behandlung komplexer Zahlen.

Die grundlegenden trigonometrischen Funktionen sind selbsterklärend, sie werden elementweise auf das Array angewendet:

```
>>> a = np.array([0, np.pi/6, np.pi/4, np.pi/3, np.pi/2])
>>> np.sin(a)
[0.         0.5        0.70710678 0.8660254  1.        ]
>>> np.cos(a)
[1.00000000e+00 8.66025404e-01 7.07106781e-01 5.00000000e-01
6.12323400e-17]
>> np.tan(a)
```

```
[0.00000000e+00, 5.77350269e-01, 1.00000000e+00,
1.73205081e+00, 1.63312394e+16]
```

Daneben ist auch die Umrechung von Radians in Grad

```
>>> a = np.array([0, np.pi/6, np.pi/4, np.pi/3, np.pi/2])
>>> np.degrees(a)
[ 0., 30., 45., 60., 90.]
```

und Grad in Radians möglich:

```
>>> a = np.array([ 0, 30, 45, 60, 90])
>>> np.radians(a)
[0.         0.52359878 0.78539816 1.04719755 1.57079633]
```

Mit `np.around` können sämtliche Werte im Array auf eine bestimmte Anzahl von Stellen gerundet werden. Ohne Angabe eines zweiten Arguments wird auf kaufmännisch auf die nächste Ganzzahl gerundet,

```
>>> a = np.array([1.49, 1.5, 1.51])
>>> np.round(a)
[1. 2. 2.]
```

mit dem optionalen zweiten Argument wird die Anzahl an Nachkommastellen, auf die gerundet werden soll, angegeben:

```
>>> a = np.array([1.25, 1.53, 1.99])
>>> np.round(a, 1)
[1.2, 1.5, 2. ]
```

Um alle Elemente eines Arrays zu aufzusummieren, kann man

```
>>> a = np.array([1, 2, 3])
>>> np.sum(a)
6
```

verwenden. Mit

```
>>> a = np.array([2, 3, 4])
>>> np.prod(a)
24
```

können die Elemente der Liste miteinander multipliziert werden.

Sollten `nan` (not a number) im Array vorkommen können, so kann `np.nansum` beziehungsweise `np.nanprod` verwendet werden. Bei `np.nansum` werden `nan` als `0` interpretiert,

```
>>> a = np.array([np.NAN, 1, 2, 3])
>>> np.sum(a)
nan
>>> np.nansum(a)
6.0
```

bei `np.nanprod` als `1`:

```
>>> a = np.array([np.NAN, 2, 3, 4])
>>> np.prod(a)
nan
>>> np.nanprod(a)
24.0
```

Das Ergebnis der Addition bzw. Multiplikation ist vom Typ `float64`. `nan` ist ein valider Fließzahlwert, daher werden die restlichen Werte in der zu konvertierenden `list` von `int` zu `float64` umgewandelt.

Kapitel 4

Weiterführende Themen

4.1 Maschinelles Lernen in Python

Das Themengebiet des maschinellen Lernens kann verschiedene Komplexitätslevel erreichen. Grundlegend ist das mathematische Verständnis über die verschiedenen im maschinellen lernen eingesetzten Algorithmen. Diese werden in diesem Tutorial nicht beschrieben.
Sind die Algorithmen bekannt und sollen nun mittels Python angewendet werden, sollte zu Beginn mit einem kleinen Projekt gestartet werden. Hierbei ist es sinnvoll sich bereits am Anfang eine Vorgehensweise zu überlegen, wie auch bei allen anderen Projekten. Python bietet im Bereich des maschinellen Lernens viele unterschiedliche Möglichkeiten an, sodass bereits frühzeitig der Aufbau des Projekts entschieden werden sollte. Grob kann ein Projekt in fünf Schritte aufgeteilt werden, anhand derer später das Ergebnis verifiziert werden kann.

1. Problem definieren
2. Daten vorbereiten
3. Algorithmen evaluieren
4. Ergebnisse verbessern
5. Ergebnisse darstellen

Eine weit verbreitete Möglichkeit ist das Einbinden von bereits existierenden Bibliotheken, die bereits gewissen Funktionalitäten von Haus aus anbieten. Eine eigene Nachbildung von verbreiteten Algorithmen aus dem Bereich Maschinelles Lernen ist daher meist nicht nötig.

Eine zweite Möglichkeit ist die Integration von R. Bei R handelt es sich um eine eigene Programmiersprache, welche den Schwerpunkt in mathematischen Problemlösungen hat. Python und R lassen sich beide sowohl eigenständig, also auch in Verbindung miteinander einsetzen.

Im Folgenden Abschnitt gibt es eine Übersicht, über wichtige und bekannte Bibliotheken aus dem Bereich maschinelles Lernen. Die Anzahl der Bibliotheken macht den Einstieg nicht ganz leicht. Die Stärken und Schwächen der einzelnen Bibliotheken sollten betrachtet werden.

4.1.1 Bibliotheken

Im Bereich maschinelles Lernen sind schon viele Bibliotheken vorhanden, die unterschiedliche Schwerpunkte in dem Bereich bedienen. Aus diesem Grund erfolgt zuerst eine Übersicht über verbreitete Bibliotheken.

Download und Installation
Alle Bibliotheken die genutzt werden sollen müssen zuerst installiert werden. Hierfür gibt es je nach Bibliothek und teilweise je nach Betriebssystem mehrere Wege. Es wird empfohlen hier aus der jeweiligen Webseite die geeignetsten Variante zu wählen und auszuführen.

Versionen und Kompatibilität
Nach der Installation sollten alle Versionen ausgelesen und abgeglichen werden. Die Kompatibilität ist nicht durchgehend gewährleistet, das betrifft vor allem die unterschiedlichen Python-Versionen.

Der folgende Codeausschnitt zeigt ein Beispiel. Dieser kann entweder direkt in der Eingabeaufforderung bzw. Konsole nach Start von Python ausgeführt werden oder innerhalb einer geeigneten Entwicklungsumgebung.

```
# Python version
import sys
print("Pyton: {}".format(sys.version))

# numpy version
import numpy
print("numpy: {}".format(numpy.__version__))

# pandas version
```

```
import pandas
print("pandas: {}".format(pandas.__version__))

# rpy2 version
import rpy2
print("rpy2: {}".format(rpy2.__version__))
```

Die Ausgabe ist beispielsweise die Folgende:

```
Pyton: 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52)
        [MSC v.1900 32 bit (Intel)]

numpy: 1.13.3
pandas: 0.21.0
rpy2: 2.8.6
```

Auf diese Art und Weise sollten alle Bibliotheken geprüft werden, da so eventuelle Kompatibilitätsprobleme oder fehlerhafte Installation frühzeitig erkannt werden können.

Bekannte Bibliotheken

Grob kann man zwischen Datenanalyse und Visualisierung unterscheiden. Zwei bekannte Bibliotheken aus dem Bereich Datenanalyse sind *numpy* und *pandas*, mit denen beispielsweise Gleichungen und Optimierungsprobleme gelöst werden können. Ergebnisse solcher Berechnungen können beispielsweise mithilfe des Moduls *matplotlib* visualisiert werden. [Kas15]

numpy
In der Bibliothek *numpy* [num] wird ein flexibler Datentyp für mehrdimensionale Arrays zur Verfügung gestellt. Dies ermöglicht eine effiziente Durchführung von komplexen Rechnungen.

Außerdem lassen sich Integrale berechnen, statistische Berechnungen durchführen und auch simulieren. All das wird für maschinelles Lernen benötigt. Da die Berechnungen mit Routinen nah an der Hardware durchgeführt werden, lassen sich bei entsprechender Programmierung effiziente Programme schreiben.

Die Arrays in *numpy* sind dreidimensional und können so eine Vielzahl von Anwendungsfällen abbilden. Der Fokus von *numpy* liegt in der Datenhal-

tung und Manipulation von Daten. Hier speziell die numerische Manipulation aus dem Bereich der linearen Algebra. Mit den Matrizen können beispielsweise Multiplikationen und Dekompositionen durchgeführt werden. Aus diesem Grund sind *numpy*-Array oft die Datenstruktur, mit der weiterführende Bibliotheken arbeiten können.

pandas
Die Webseite zu *pandas* [pan] beschreibt dieses als gute Wahl zur schnellen und flexiblen Aufbereitung von Daten. *pandas* bietet verschiedene Möglichkeiten, um schnell auf Einträge zuzugreifen. Dies ist möglich, da mittels *pandas* Serien und Dataframes erzeugt werden können, die im Gegensatz zu einem Array in Python auch Spaltentitel und Indizes können.

Die *describe()*-Methode bietet hierbei einen ersten Überblick über die im Dataframe enthalten Daten. Ohne weitere Programmierung werden Informationen wie Maximalwert, Minimalwert und Durchschnitt für jede Spalte berechnet und angezeigt.

Spalten und Zeilen können mittels *pandas* gefiltert, erweitert und verändert werden, sodass Pandas oft im ersten Schritt genutzt wird, um die auszuwertenden Daten zu laden und genauer analysieren zu können.

scipy
Ergänzend bzw. aufbauen auf *numpy* werden durch *scipy* [scib] viele mathematische Operationen bereit gestellt. Das Modul *scipy* ist sehr mächtig und daher nochmal in Untermodule aufgeteilt. Innerhalb der Untermodule werden bestimmte Funktionalitäten gruppiert. Eine Übersicht hierzu gibt die Onlinedokumentation.

scikit-learn
Viele DataMining bzw. Machine Learning Funktionalitäten werden bereits durch die *scikit-learn* [scia] Bibliothek (manchmal aus sklearn abgekürzt) zur Verfügung gestellt. Die klassischen Algorithmen, wie *k-Means* oder *knn* sind bereits integriert.
Des Weiteren ist auch mit *scikit-learn* eine Aufbereitung der Daten möglich. Hier werden beispielsweise Normalierung und Skalierung unterstützt. Insgesamt handelt es sich um eine sehr mächtige Bibliothek, mit der es möglich ist unterschiedliche Algorithmen zu probieren und verschiedene Test- und Trainingsverfahren zu testen. Es können auch neue Daten anhand der gelernten Modelle klassifiziert und vorhergesagt werden.

rpy2
Eine weitere Möglichkeit ist das Einbinden des Pakets *rpy2* [rpy]. Hierbei handelt es sich um eine Bibliothek, welche Komponenten aus R zur Verfügung stellt. *rpy2* ist zu sehen wie eine Schnittstelle zwischen Python und R. Es ist möglich R-Pakete mittels Python zu importieren und mit den darin enthaltenen Funktionalitäten zu interagieren.

matplotlib
Mit dem Modul *matplotlib* [mat] können Daten in einem Diagramm dargestellt werden. Hiermit kann ein erstes Verständnis der Daten oder Ergebnisse erreicht werden. Es werden unter anderem Liniendiagramme, Histogramme, Balkendiagramme aber auch Heatmaps unterstützt. Hier können sowohl Achsen, Farben und auch Beschriftungen nach Bedarf angepasst werden. *matplotlib* unterstützt sowohl *pandas* als auch *numpy* und ist daher oft bereits am Anfang von hoher Bedeutung, um einen Überblick über die Daten zu bekommen.

Literaturverzeichnis

[Kas15] KASIER, ERNESTI: *Python 3 Das umfassende Handbuch*. Rheinwerk VerlagGBmbH, 2015.

[mat] *matplotlib.org*. https://matplotlib.org/. zuletzt gesehen am 24.10.2018.

[num] *numpyreference*. https://docs.scipy.org/doc/numpy/reference/index.html. zuletzt gesehen am 24.10.2018.

[pan] *pandas*. http://pandas.pydata.org/pandas-docs/stable/. zuletzt gesehen am 24.10.2018.

[rpy] *rpy2*. https://pypi.org/project/rpy2/. zuletzt gesehen am 01.11.2018.

[scia] *scikit-learn*. http://scikit-learn.org/stable/index.html. zuletzt gesehen am 01.11.2018.

[scib] *scipyreference*. https://docs.scipy.org/doc/scipy/reference/. zuletzt gesehen am 24.10.2018.